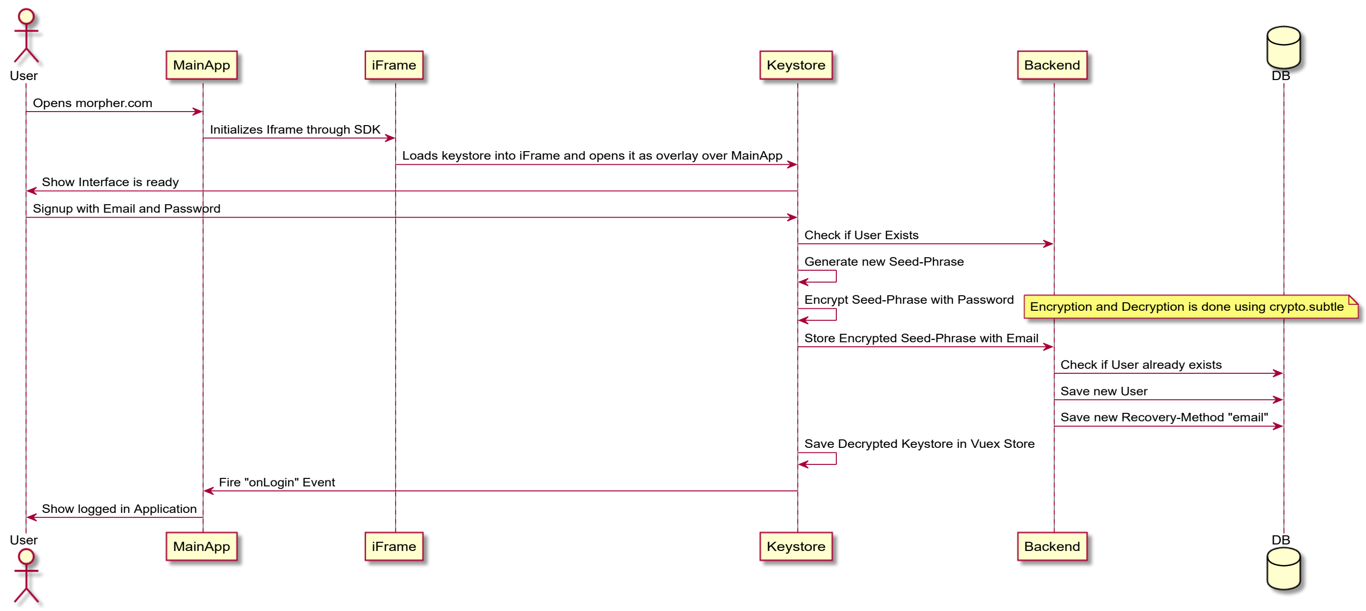
Morpher Wallet

This is the guideline for a security review of the Morpher Wallet

General thoughts for the wallet

Getting started quickly

Iframe Flow

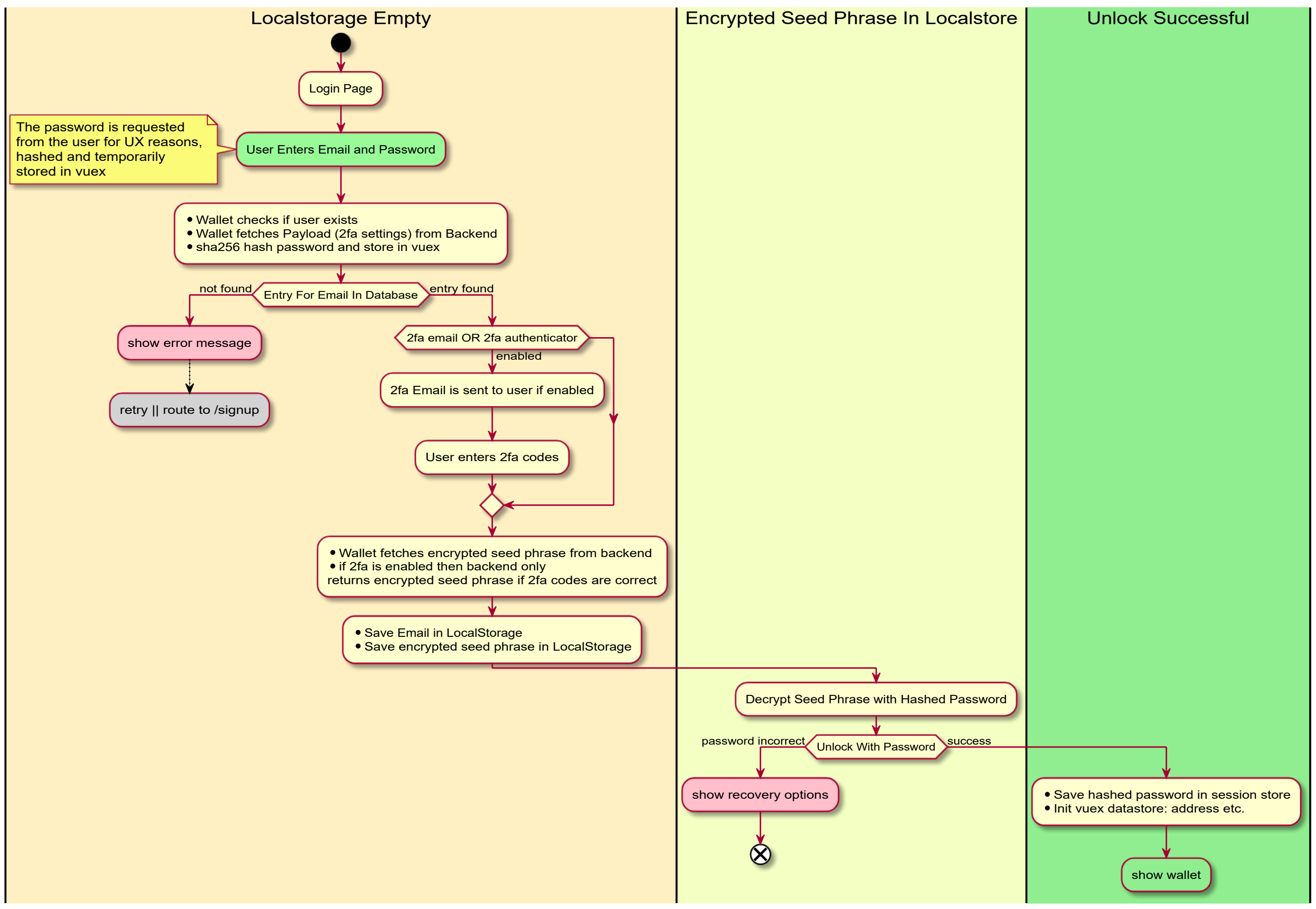

User
MainApp
iFrame
Keystore
Backend
DB
Opens morpher.com
Initializes Iframe through SDK
Loads keystore into iFrame and opens it as overlay over MainApp
Show Interface is ready
Signup with Email and Password
Check if User Exists
Generate new Seed-Phrase
Encrypt Seed-Phrase with Password
Encryption and Decryption is done using crypto.subtle
Store Encrypted Seed-Phrase with Email
Check if User already exists
Save new User
Save new Recovery-Method "email"
Save Decrypted Keystore in Vuex Store
Fire "onLogin" Event
Show logged in Application

Login Flow for the Wallet

This section describes the login and unlock-Flow for the wallet.

The login and unlock flow are different only by the contents of the localStorage. If the encryptedSeed is already stored in localStorage, then we don't need to fetch it from the backend.

Login Flow


Localstorage Empty
Encrypted Seed Phrase In Localstore
Unlock Successful
Login Page
The password is requested from the user for UX reasons, hashed and temporarily stored in vuex
User Enters Email and Password
• Wallet checks if user exists
• Wallet fetches Payload (2fa settings) from Backend
• sha256 hash password and store in vuex
not found
Entry For Email In Database
entry found
show error message
retry || route to /signup
2fa email OR 2fa authenticator
enabled
2fa Email is sent to user if enabled
User enters 2fa codes
• Wallet fetches encrypted seed phrase from backend
• if 2fa is enabled then backend only returns encrypted seed phrase if 2fa codes are correct
• Save Email in LocalStorage
• Save encrypted seed phrase in LocalStorage
Decrypt Seed Phrase with Hashed Password
password incorrect
Unlock With Password
success
show recovery options
• Save hashed password in session store
• Init vuex datastore: address etc.
show wallet

When the user logs into the wallet the first time, then the browsers localstore and sessionstore is empty.

1. The user is required to enter Email-Address and Password.

   The password is temporarily stored, because of UX reasons. It is used later in the process.

2. The Wallet checks with the backend if the user already exists.

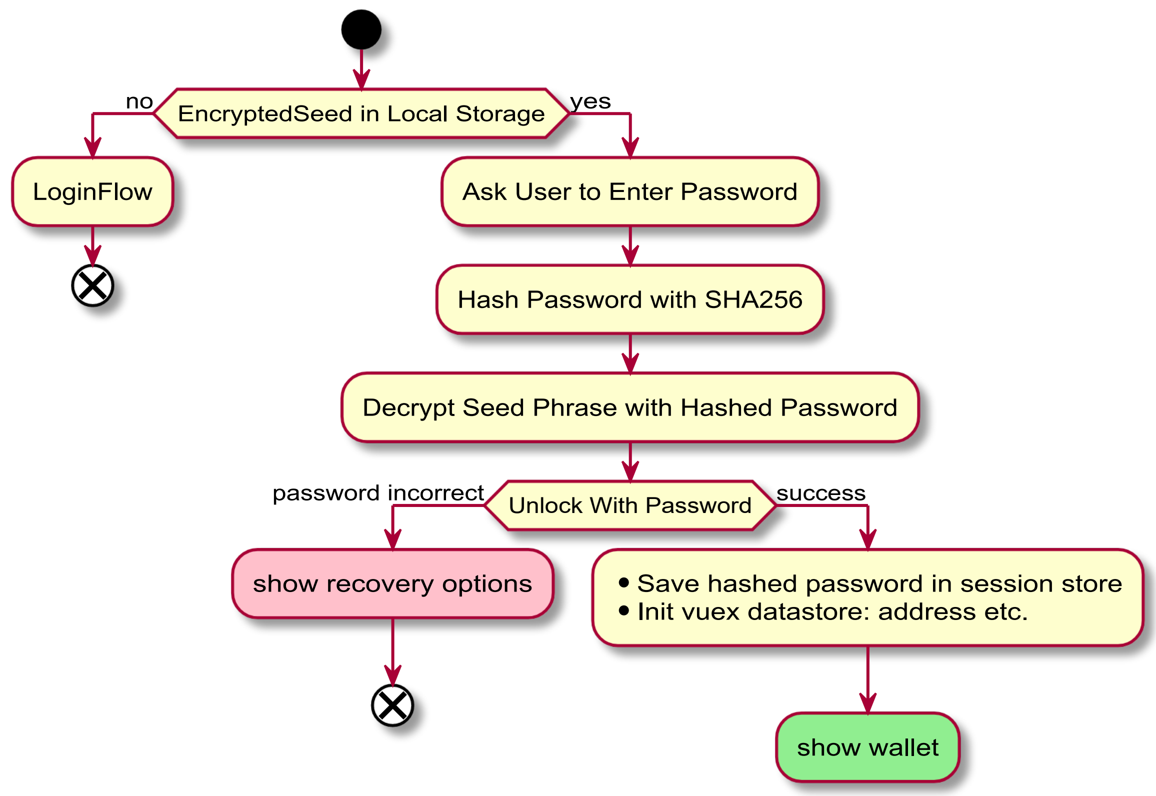
   Sidenote: It became "good behavior" to show generic error messages to avoid leaking user-exists information. This simply doesn't make sense, because the backend is a data-store for an encrypted wallet. The actual decryption process is done in-browser.

3. Depending on the 2FA settings, the user is then required to enter the 2FA codes. The backend encrypts the wallet with the 2fa codes and only the correct codes + password can decrypt the keystore.

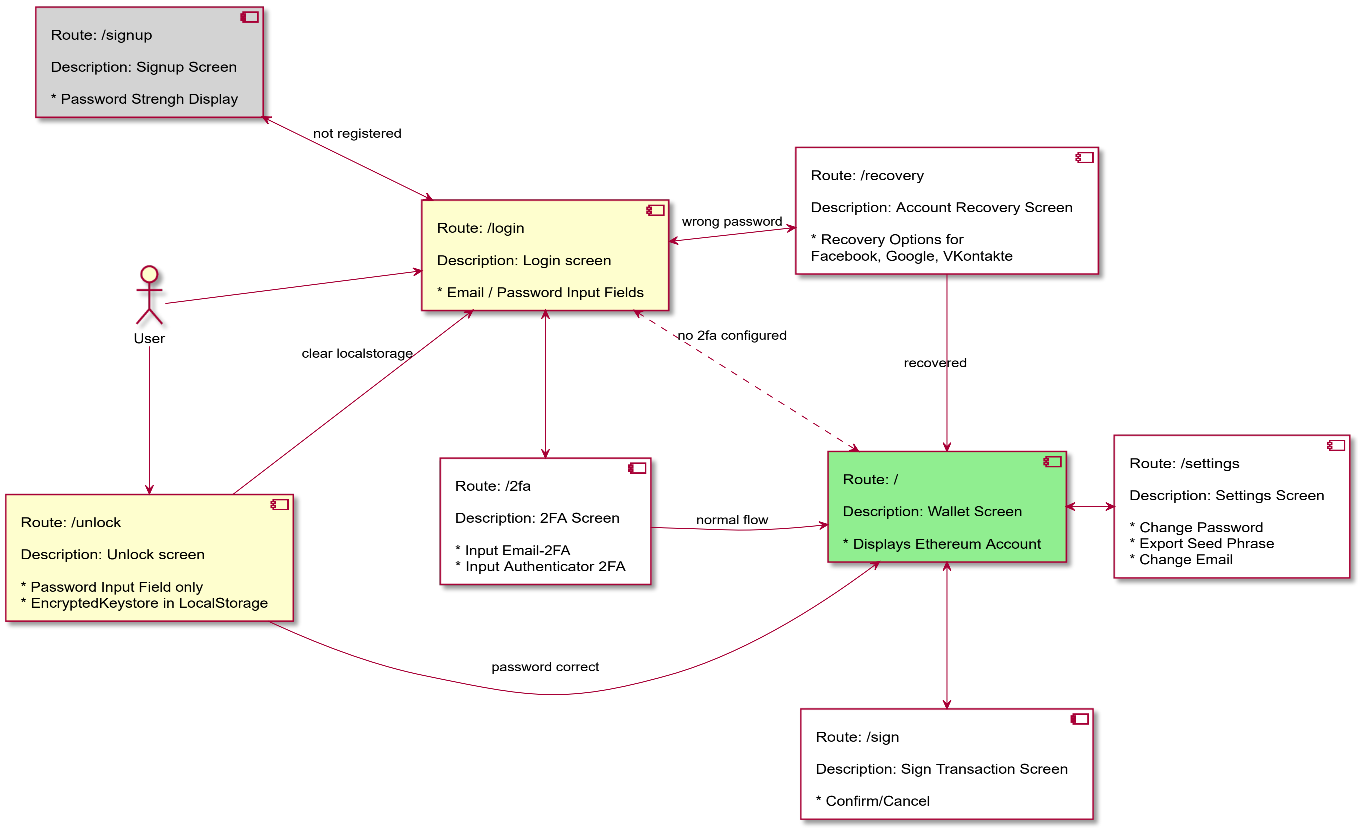
4. If decryption with 2FA and password was successful then the *password-encrypted* keystore (after decrypt with 2fa) is stored in localstorage. This lets the user "unlock" the keystore with the password only in sub-sequent uses of the wallet

5. The wallet is initialized: The keystore is used to generate an account. The account is stored in the JavaScript-Storage-Backend. The frontend shows the Address and UI

Unlock Flow


EncryptedSeed in Local Storage
no
yes
LoginFlow
Ask User to Enter Password
Hash Password with SHA256
Decrypt Seed Phrase with Hashed Password
Unlock With Password
password incorrect
success
show recovery options
• Save hashed password in session store
• Init vuex datastore: address etc.
show wallet

Description of the Routes


Route: /signup
Description: Signup Screen
* Password Strengh Display
not registered
Route: /login
Description: Login screen
* Email / Password Input Fields
wrong password
Route: /recovery
Description: Account Recovery Screen
* Recovery Options for Facebook, Google, VKontakte
User
clear localstorage
no 2fa configured
recovered
Route: /unlock
Description: Unlock screen
* Password Input Field only
* EncryptedKeystore in LocalStorage
Route: /2fa
Description: 2FA Screen
* Input Email-2FA
* Input Authenticator 2FA
normal flow
Route: /
Description: Wallet Screen
* Displays Ethereum Account
Route: /settings
Description: Settings Screen
* Change Password
* Export Seed Phrase
* Change Email
password correct
Route: /sign
Description: Sign Transaction Screen
* Confirm/Cancel

Route	description	Meta-Information	Next Screens Possible
Login: /login	The first screen if the LocalStorage is empty	encryptedKeystoreInStorage == false && keystoreUnlocked == false	Signup, Recovery, 2fa, Wallet
Signup: /signup	If the user wants to create a new Account, then this is the component which takes care of that.	encryptedKeystoreInStorage == false && keystoreUnlocked == false	Login, 2fa
Unlock: /unlock	Let's the user unlock the keystore	encryptedKeystoreInStorage == true && keystoreUnlocked == false	
2FA: /twoFa	If the user is required to enter 2fa codes, then he is redirected here	mail2fa or authenticator2fa = true	Wallet, Login
Wallet: /	If the user is logged in correctly, he will see the wallet address and can switch accounts	keystoreUnlocked = true	Login, Settings, Sign
Settings: /settings	Change Email, Password, Export Seed Phrases, Add Account Recovery	keystoreUnlocked = true	Wallet

Route	description	Meta-Information	Next Screens Possible
Sign Transaction: /signtx	Shows a summary of the transaction and a sign/cancel button	keystoreUnlocked = true	Wallet

Overview

This document describes the external packages used in the Morpher Wallet.

Frontend Technologies

The frontend is split into two parts:

1. The Keystore
2. The iFrame SDK

In general both are written in TypeScript (or ported to TS).

The Keystore is written in VueJs with a Vuex datastore.

The iFrame SDK is written using a native iFrame HTML element and binding parent/child listeners through penpal to the parent/child document. More on the flow below.

List of Tools/Frameworks Used

The following is a non-exhaustive list of important packages used in the Keystore-App with their description:

Web3js Wallet

https://web3js.readthedocs.io/en/v1.2.0/web3-eth-accounts.html
The web3.eth.accounts contains functions to generate Ethereum accounts and sign transactions and data.

It's currently marked as "*This package has NOT been audited and might potentially be unsafe. Take precautions to clear memory properly, store the private keys safely, and test transaction receiving and sending functionality properly before using in production!*". We're aware of this fact, no data leakage is indicated at this point. Web3js is a well maintained open source library and widely in use. Web3.eth.accounts is giving back an object containing scoped functions to sign a transaction. The object is never exposed

```
{
    address: "0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01",
    privateKey:
"0x348ce564d427a3311b6536bbcff9390d69395b06ed6c486954e971d960fe8709",
    signTransaction: function(tx){...},
    sign: function(data){...},
    encrypt: function(password){...}
}
```

Web3js can either import private keys or generate them. Since we base the wallet on *seed-phrases*, another library, bip39, is used to generate the seed phrases.

bip39

https://www.npmjs.com/package/bip39

JavaScript implementation of Bitcoin BIP39: Mnemonic code for generating deterministic keys

```
const mnemonic = bip39.generateMnemonic()
// => 'seed sock milk update focus rotate barely fade car face mechanic mercy'
```

This generates a mnemonic using crypto.getRandomValues in the browser and then fetches words from an english wordlist. The wordlist is 2048 words long and the entropy generated by default is 128 bits. This gives 16 random bytes using the randombytes package.

This mnemonic is then piped through ethereumjs-utils to generate a private key.

hdkey from ethereumjs-utils

https://www.npmjs.com/package/ethereumjs-util is a collection of utility functions for Ethereum.

The Morpher Wallet uses the hdkey functions from ethereumjs-wallet

We use it in vue/src/utils/keystore.ts to generate the private keys. Sample code:

```
import { hdkey } from 'ethereumjs-wallet';
function getPrivateKeyFromMnemonic(mnemonic: string, index: number) {

    const seed = mnemonicToSeedSync(mnemonic);
    const hdwallet = hdkey.fromMasterSeed(seed);
    const walletHdPath = "m/44'/60'/0'/0/";

    const wallet = hdwallet.derivePath(walletHdPath + index).getWallet();

    const privateKey = wallet.getPrivateKey().toString("hex");

    return privateKey;
}
```

Wallet Settings

This describes the flow for the settings.

Middleware

All writing operations for user-settings are bound to a middleware. All requests are signed using the private key of the user as secret and the complete request-payload including a one-time nonce as message content.

Request Signing

To authenticate a stateless request to the backend to update user information, the requests are signed.

The signature method is very similar to HMAC signatures used by Amazon.

In general it prevents:

1. MIDM Attacks by signing the payload
2. Replay Attacks by introducing a nonce

Nonce

The nonce is a known number that increases every time an authenticated request is sent to the backend. No nonce is ever the same. The nonce is part of the payload.

Payload

When the payload for POST or GET requests is sent, then an additional header is added to the request with the signature.

Signature

The signature is generated by using the private key of the user. The eth-address for the user is stored backend-side in the database. In general, every authenticated request must fulfill the following equation:

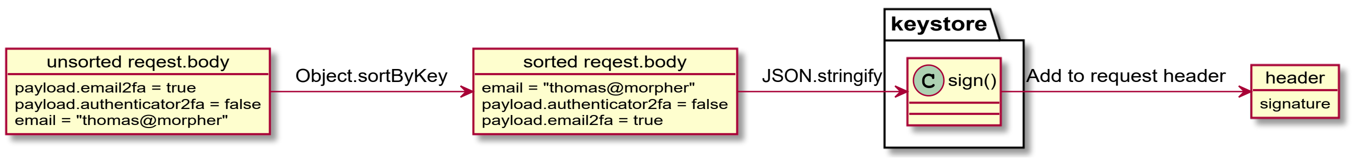
`eth_address_stored_in_backend == ec_recover(request.headers.signature, request.body.payload);`

Otherwise it will fail.

To recover the eth_address we use https://web3js.readthedocs.io/en/v1.2.0/web3-eth-personal.html#ecrecover

Client Side Signature Creation

The signature is created client-side with the keystore (web3.eth.accounts.wallet.sign).
In general it follows this flow:



The following pseudo-code is responsible for the signature creation:

```
function addSignature(requestObject) {
    requestObject.body.nonce = getNonceFromBackend();

    /**
    requestObject.body e.g.:
    {
        email: "thomas@morpher.some",
        payload: {
            email2fa: true,
            authenticator2fa: false
        },
        nonce: 12345
    }
    */
    data_sorted = sortAlphabeticallyByKey(requestObject.body)
    //watch out that also sub-objects are sorted! So it looks like this:
    data_sorted = {
        email: ...,
        nonce: ...,
        payload: {
            authenticator2fa: ...,
            email2fa: ...
        }
    }

    //then stringify it - either like this:
    signedMessage = keystore.sign(JSON.stringify(data_sorted))

    requestObject.headers.signedMessage = signedMessage;
    requestObject.headers.signatureKey = sha256(email);
    return requestObject
}

//here the function calls the backend and adds the signature
fetch('/backendUrl', addSignature(requestObject));
```

Backend

On the backend a middleware is checking the request body against the users eth_address after ec_recovery of the signature:

In Pseudocode:

```
recovery = findOne({
        where: {
            "key": req.headers.signatureKey
        }
    }, include: User)
if (recovery) {
    eth_address = ecrecover(JSON.stringify(sortAlphabetically(req.body)),
req.headers.signedMessage)
    if (eth_address == recovery.user.eth_address) {
        //SUCCESS HERE
        //routing continues
    }

}

throw 505-error //happens in any other case.
```

Password Change

To change the password, the user has to provide the old password and repeat the new password twice.

The same validation logic as in signup applies to the password change mechanism. It must fulfill:

1. Minimum length
2. Type of characters
3. Combination of upper/lower case

If the validation passes then the users

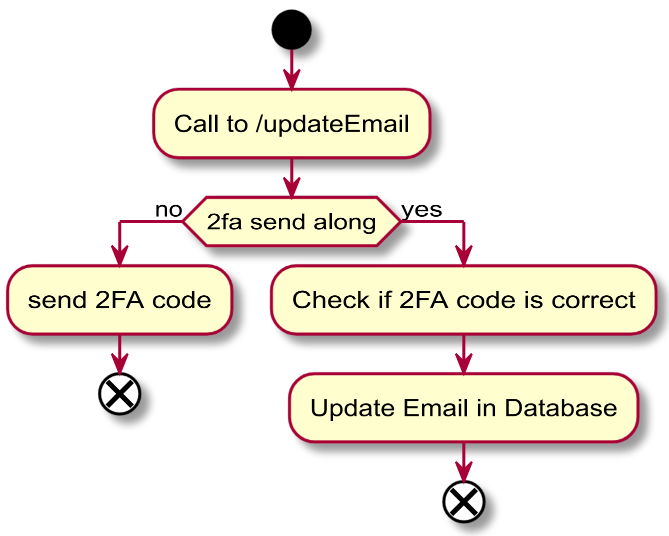
1. keystore is decrypted using the old-password.
2. re-encrypted using the new password
3. the encrypted keystore is sent to the server to replace the old keystore
4. server-side the keystore is encrypted with the server-side keys

Email Change

Changing the Email Address is a more complex topic than it appears on the surface.
Pre-assumptions:

- The user has successfully unlocked his keystore and has an unlocked wallet in localstorage.
- The user might enter a wrong *new* email address, which needs to be validated.

Possible attack vector: Someone having access to the users computer might maliciously try to change the users email address.



On the first request a 2FA code is generated and sent to the *new* email address.

The user is then requested to enter the 2FA code - to validate that he really is the owner of the new email address - and send it along with the payload.

Note: It is clear that writing a different, un-validated, new email-address potentially *can* be circumvented on a lower level by grabbing a 2fa code from the first request and change the target email address on the subsequent request. The additional layers are for ux reasons and sanity-checks, rather than bullet-proof security. As soon as the user is in possession of an unencrypted keystore it is to be expected that changing an email address is the least of the problems and the intention of the user.